SMART CONTRACT

Security Audit Report

Project: Quant Token
Website: quant.network
Platform: Ethereum
Language: Solidity
Date: May 14th, 2024

Table of contents

`

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

Introduction

As part of EtherAuthority's community smart contracts audit initiatives, the smart contracts of Quant Token from quant.network were audited. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on May 14th, 2024.

The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contract.

Project Background

- This Solidity contract implements a standard ERC20 token with additional functionality for a crowdsale. Let's break it down:
  - **ERC20Basic:**
    - **Interface defining essential ERC20 functions:** `totalSupply()`, `balanceOf()`, and `transfer()`.
    - It emits a `Transfer` event upon successful token transfer.
  - **SafeMath Library:**
    - A library providing arithmetic functions with safety checks to prevent overflow and underflow.
  - **BasicToken:**
    - Inherits from ERC20Basic and uses SafeMath library.
    - Implements the `totalSupply()` and `balanceOf()` functions.
    - Implements the `transfer()` function with safety checks for valid addresses and sufficient balance.
    - The total supply is initialized to a fixed value of 45,467,000 QNT tokens.
  - **ERC20 Interface:**
    - **An interface extending ERC20Basic with additional functions:** `allowance()`, `transferFrom()`, and `approve()`.
    - It emits an `Approval` event upon successful approval.
  - **StandardToken:**

- Inherits from ERC20 and BasicToken.
- Implements additional functionalities such as `approve()`, `allowance()`, `transferFrom()`, `increaseApproval()`, and `decreaseApproval()`.
- Defines constants for token name, symbol, and decimals.
- Defines a `crowdsale` address and a modifier `onlyCrowdsale` to restrict certain functions to the crowdsale contract.
- Implements a `mint()` function that allows the crowdsale contract to mint tokens to specific addresses.
- Overrides the `transferFrom()` function with additional safety checks for valid addresses, sufficient balance, and allowance.
- Implements the `approve()` function to allow the spender to spend the specified amount on behalf of the owner.
- Implements `increaseApproval()` and `decreaseApproval()` functions to atomically increase or decrease the allowance.

- Overall, this contract provides a standard ERC20 token functionality with additional features such as minting and allowance manipulation, along with safety checks to prevent common vulnerabilities like overflow and underflow.

Audit scope

Name	Code Review and Security Analysis Report for Quant Token Smart Contract
Platform	Ethereum
File	StandardToken.sol
Smart Contract Code	0x4a220e6096b25eadb88358cb44068a3248254675
Audit Date	May 14th, 2024

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: ● Name: Quant ● Symbol: QNT ● Decimals: 18 ● Decimal Zeros: 18	**YES, This is valid.**
Crowdsale control: ● Mint new tokens by the Crowdsale owner.	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, Customer`s solidity based smart contracts are **"Secured"**. This token contract does not have any ownership control, hence it is 100% decentralized.

Insecure	Poor secured	Secure	Well-secured

You are here ➜

We used various tools like Slither, Solhint and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.

All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit overview section. General overview is presented in AS-IS section and all identified issues can be found in the Audit overview section.

We found 0 critical, 0 high, 0 medium and 0 low and 7 very low level issues.

Investors Advice: Technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

Technical Quick Stats

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Features claimed	Passed
	Other programming issues	Moderated
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Passed
Gas Optimization	"Out of Gas" Issue	Passed
	High consumption 'for/while' loop	Passed
	High consumption 'storage' storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage not set	Moderated
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

Overall Audit Result: PASSED

This is a private and confidential document. No part of this document should be disclosed to third party without prior written permission of EtherAuthority.

Email: audit@EtherAuthority.io

Business Risk Analysis

Category	Result
🟢 Buy Tax	0%
🟢 Sell Tax	0%
🟢 Cannot Buy	No
🟢 Cannot Sell	No
🟢 Max Tax	0%
🟢 Modify Tax	Not Detected
🟢 Fee Check	No
🟢 Is Honeypot	Not Detected
🟢 Trading Cooldown	Not Detected
🟢 Can Pause Trade?	No
🟢 Pause Transfer?	Not Detected
🟢 Max Tax?	No
🟢 Is it Anti-whale?	Not Detected
🟢 Is Anti-bot?	Not Detected
🟢 Is it a Blacklist?	Not Detected
🟢 Blacklist Check	No
🟢 Can Mint?	Yes
🟢 Is it Proxy?	No
🟢 Can Take Ownership?	No
🟢 Hidden Owner?	No
🟢 Self Destruction?	Not Detected
🟢 Auditor Confidence	High

Overall Audit Result: PASSED

Code Quality

This audit scope has 1 smart contract. Smart contract contains Libraries, Smart contracts, inherits and Interfaces. This is a compact and well written smart contract.

The libraries in Quant Token are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in the Quant Token.

The EtherAuthority team has no scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Code parts are well commented on in the smart contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a Quant Token smart contract code in the form of an [Etherscan](#) web link.

As mentioned above, code parts are well commented on. and the logic is straightforward. So it is easy to quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are based on well known industry standard open source projects.

Apart from libraries, its functions are not used in external smart contract calls.

AS-IS overview

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	onlyCrowdsale	modifier	Missing error message	Refer Audit Findings
3	StandardToken	write	Passed	No Issue
4	mint	write	Centralized risk, Unlimited Quant tokens minting, Public Functions Should be Declared External	Refer Audit Findings
5	transferFrom	write	Missing error message, Public Functions Should be Declared External	Refer Audit Findings
6	approve	write	Public Functions Should be Declared External	Refer Audit Findings
7	allowance	read	Public Functions Should be Declared External	Refer Audit Findings
8	increaseApproval	write	Public Functions Should be Declared External	Refer Audit Findings
9	decreaseApproval	write	Public Functions Should be Declared External	Refer Audit Findings
10	allowance	read	Passed	No Issue
11	transferFrom	write	Passed	No Issue
12	approve	write	Passed	No Issue
13	totalSupply	read	Passed	No Issue
14	transfer	write	Missing error message	Refer Audit Findings
15	balanceOf	read	Passed	No Issue
16	totalSupply	read	Passed	No Issue
17	balanceOf	read	Passed	No Issue
18	transfer	write	Passed	No Issue

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

No low severity vulnerabilities were found.

Very Low / Informational / Best practices:

(1) Use latest solidity version:

```
5    pragma solidity ^0.4.21;
```

Use the latest solidity version while contract deployment to prevent any compiler version level bugs.

Resolution: Please use versions greater than 0.8.7.

(2) Explicit Visibility for State Variables Warning:

```
mapping(address => uint256) balances;

uint256 totalSupply_ = 4546700000000000000000000000;
```

The warning is related to the visibility of state variables in your Solidity code.

Resolution: We recommend updating the code to explicitly mark the visibility of state variables using the internal or public keyword, depending on the intended visibility.

(3) Missing error message:

```solidity
function transferFrom(address _from, address _to, uint256 _value) public returns (bool)
  require(_to != address(0));
  require(_value <= balances[_from]);
  require(_value <= allowed[_from][msg.sender]);

  balances[_from] = balances[_from].sub(_value);
  balances[_to] = balances[_to].add(_value);
  allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
  emit Transfer(_from, _to, _value);
  return true;
}
```

```solidity
function transfer(address _to, uint256 _value) public returns (bool) {
  require(_to != address(0));
  require(_value <= balances[msg.sender]);

  // SafeMath.sub will throw if there is not enough balance.
  balances[msg.sender] = balances[msg.sender].sub(_value);
  balances[_to] = balances[_to].add(_value);
  emit Transfer(msg.sender, _to, _value);
  return true;
}
```

```solidity
modifier onlyCrowdsale() {
  require(msg.sender == crowdsale);
  _;
}
```

There is no error message set in the required condition.

Resolution: We suggest setting relevant error messages to identify the failure of the transaction.

(4) Centralized risk:

```solidity
function mint(address _address, uint256 _value) public onlyCrowdsale {
  balances[_address] = balances[_address].add(_value);
  emit Transfer(0, _address, _value);
}
```

Only onlyCrowdsale can mint a token.

Resolution: To make the smart contract 100% decentralized. We suggest renouncing ownership of the smart contract once its function is completed.

(5) Make variables constant:

```
uint256 totalSupply_ = 4546700000000000000000000000;
```

These variable's values will be unchanged. So, please make it constant. It will save some gas. Just put a constant keyword.

(6) Unlimited Quant tokens minting:

```
function mint(address _address, uint256 _value) public onlyCrowdsale {
  balances[_address] = balances[_address].add(_value);
  emit Transfer(0, _address, _value);
}
```

The owner of this contract can mint unlimited tokens. Having unlimited minting is considered inappropriate for tokenomics. So, we suggest either putting some minting limits, including permission checks and authorization for minting functions.

(7) Public Functions Should be Declared External:

The below functions in this contract should be declared as external in order to save gas.

- mint
- transferFrom
- approve
- allowance
- increaseApproval
- decreaseApproval

Resolution: Improve gas efficiency by declaring suitable public functions as external. Enhance contract performance while optimizing gas usage.

Centralization Risk

The Quant smart contract does not have any ownership control, **hence it is 100% decentralized.**

Therefore, there is **no** centralization risk.

Conclusion

We were given a contract code in the form of Etherscan web links. And we have used all possible tests based on given objects as files. We had observed 7 informational issues in the smart contracts. And those issues are not critical. So, **it's good to go for the production**.

Since possible test cases can be unlimited for such smart contracts protocol, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in the As-is overview section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed smart contract, based on standard audit procedure scope, is **"Secured".**

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

Appendix

Code Flow Diagram - Quant Token
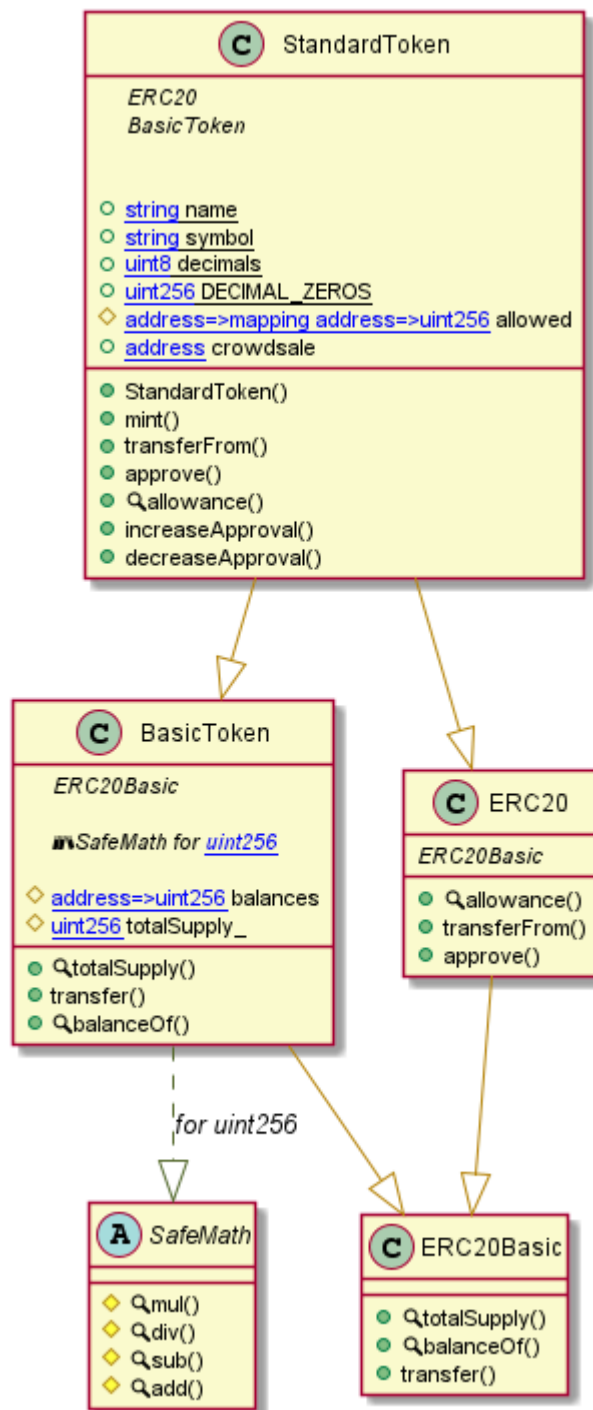
Slither Results Log

Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We did the analysis of the project altogether. Below are the results.

Slither Log >> StandardToken.sol

```
SafeMath.div(uint256,uint256) (StandardToken.sol#40-45) is never used and should be removed
SafeMath.mul(uint256,uint256) (StandardToken.sol#28-35) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

Pragma version^0.4.24 (StandardToken.sol#5) allows old versions
solc-0.4.24 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Parameter BasicToken.transfer(address,uint256)._to (StandardToken.sol#88) is not in mixedCase
Parameter BasicToken.transfer(address,uint256)._value (StandardToken.sol#88) is not in mixedCase
Parameter BasicToken.balanceOf(address)._owner (StandardToken.sol#104) is not in mixedCase
Parameter StandardToken.mint(address,uint256)._address (StandardToken.sol#153) is not in mixedCase
Parameter StandardToken.mint(address,uint256)._value (StandardToken.sol#153) is not in mixedCase
Parameter StandardToken.transferFrom(address,address,uint256)._from (StandardToken.sol#164) is not in mixedCase
Parameter StandardToken.transferFrom(address,address,uint256)._to (StandardToken.sol#164) is not in mixedCase
Parameter StandardToken.transferFrom(address,address,uint256)._value (StandardToken.sol#164) is not in mixedCase
Parameter StandardToken.approve(address,uint256)._spender (StandardToken.sol#186) is not in mixedCase
Parameter StandardToken.approve(address,uint256)._value (StandardToken.sol#186) is not in mixedCase
Parameter StandardToken.allowance(address,address)._owner (StandardToken.sol#198) is not in mixedCase
Parameter StandardToken.allowance(address,address)._spender (StandardToken.sol#198) is not in mixedCase
Parameter StandardToken.increaseApproval(address,uint256)._spender (StandardToken.sol#212) is not in mixedCase
Parameter StandardToken.increaseApproval(address,uint256)._addedValue (StandardToken.sol#212) is not in mixedCase
Parameter StandardToken.decreaseApproval(address,uint256)._spender (StandardToken.sol#228) is not in mixedCase
Parameter StandardToken.decreaseApproval(address,uint256)._subtractedValue (StandardToken.sol#228) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
```

```
StandardToken.slitherConstructorVariables() (StandardToken.sol#128-240) uses literals with too many digits:
        - totalSupply_ = 4546700000000000000000000 (StandardToken.sol#74)
StandardToken.slitherConstructorConstantVariables() (StandardToken.sol#128-240) uses literals with too many digits:
        - DECIMAL_ZEROS = 1000000000000000000 (StandardToken.sol#137)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits

BasicToken.totalSupply_ (StandardToken.sol#74) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant
StandardToken.sol analyzed (5 contracts with 84 detectors), 23 result(s) found
```

Solidity Static Analysis

Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

StandardToken.sol

Gas costs:

Gas requirement of function StandardToken.transfer is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 88:2:

Gas costs:

Gas requirement of function StandardToken.transfer is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 88:2:

Similar variable names:

BasicToken.balanceOf(address) : Variables have very similar names "balance" and "balances". Note: Modifiers are currently not considered by this static analysis.
Pos: 105:11:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
more
Pos: 167:4:

Solhint Linter

Linters are the utility tools that analyze the given source code and report programming errors, bugs, and stylistic errors. For the Solidity language, there are some linter tools available that a developer can use to improve the quality of their Solidity contracts.

StandardToken.sol

```
Compiler version ^0.4.21 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:4
Explicitly mark visibility of state
Pos: 3:71
Explicitly mark visibility of state
Pos: 3:73
Provide an error message for require
Pos: 5:88
Provide an error message for require
Pos: 5:89
Constant name must be in capitalized SNAKE_CASE
Pos: 3:130
Constant name must be in capitalized SNAKE_CASE
Pos: 3:132
Constant name must be in capitalized SNAKE_CASE
Pos: 3:134
Provide an error message for require
Pos: 5:143
Provide an error message for require
Pos: 5:148
Provide an error message for require
Pos: 5:164
Provide an error message for require
Pos: 5:165
Provide an error message for require
Pos: 5:166
```

Software analysis result:

These software reported many false positive results and some are informational issues. So, those issues can be safely ignored.